Project 1: Concurrent Web Server using BSD Sockets Report

CS 118: Computer Networking Fundamentals

Matthew Lin
904281426

February 5, 2015
TA: Seungbae Kim

High-level description of my system's design:
The system design relies on TCP. To run the system, I used two terminals: one to represent the client-side, the other to represent the server-side. Because I was running both sides on the same machine, I ran the ports via localhost rather than specifying the full IP address for

convenience (less typing and I don't need to memorize the full IP address). I used the sample skeleton code for client.c and server.c from CCLE. The only changes to the code I made were in server.c. I implemented forking and terminating zombie processes because this would allow the server to run indefinitely until the user hits “CTRL+C”. I added a function called parse that takes in an integer whose value is determined through the accept function, which allows us to connect to the server. The parse function performs the HTTP request by looking for the headers. If it does not see a valid file, it will throw an error. Otherwise, it tries to find the proper file extension. Once we know the file type, the server writes the content-length and the content-type. I created a dynamically allocated character pointer to load the file into memory, and I write out the loaded file and close it upon completion.

Difficulties that I encountered:
Overall, the hardest part for me was to figure out exactly what the spec wanted. The wording of the spec was a little confusing because I thought I had to write code for Part A, and I had to write only a small function for Part B. Upon clarifying with Seungbae more in office hours, however, I eventually learned that the project entailed very minimal work on Part A and a lot more work on Part B.

In terms of implementation, I had the most trouble implementing the forking. This was a topic I particularly struggled with in CS 111 and I was experiencing a lot of slowdowns in my virtual machine as well as runtime errors since I was not forking correctly. Eventually, I was able to implement forking, and the reason forking is of great importance is because you can run it indefinitely and it can handle simultaneous connections, which is usually what servers in the real-world do.

The second significant challenge I faced was allocating the size of the nbytes while I was reading into my buffer. Originally, I put an arbitrary amount of bytes to allocate to my buffer (210 bytes), but this caused my server to stall midway and not output the entire message. I made the buffer larger (512 bytes), and I read in sizeof(buffer) in order to output the full message without problems.

The third major difficulty I faced involved the writes when I was working with .gif and .jpg files. Originally, I was having trouble implementing the file extensions because I did not allocate the correct size, so this caused some issues where I was outputting random Greek characters. It turns out I was not allocating enough bytes for each character message because I was using sizeof(buffer) instead of the size of the string message. Upon making this fix, I was able to send the .gif and .jpg files correctly.

How to compile and run my source code:
1. Enter “make” into Terminal. This compiles the server.c and server.o files and generates an executable file called server. The Makefile produces the exact same executable as one you would create if you ran the command “gcc -o server server.c”.
2. Inside the code, you have the option of running either Part A or Part B. These are lines 26 and 29 for Part A and Part B, respectively. Uncomment the option you want to use, then proceed through the following instructions.

Part A:
I ran the following sequence of instructions:

1. Run “./server 10000” on a terminal
2. Run “./client 192.168.0.151 10000” on a different terminal
3. Run “localhost:10000/” on a browser (I used Firefox)
4. On the browser screen, I saw the text “I got your message” for about 1 second
5. Check the terminal window where I ran “./server 10000”. The message is below in the Sample outputs section.

Part B:
I ran this sequence of instructions:
1. Run “./server 10001” on a terminal
2. Run “.client 192.168.0.151 10001” on a different terminal
3. Run “localhost:10001/test.txt” on a browser (I used firefox)
* Outputs a text file into the browser

4. Run “./server 10002” on a terminal
5. Run “.client 192.168.0.151 10002” on a different terminal
6. Run “localhost:10002/dog.gif” on a browser (I used firefox)
* Plays a GIF file of a dog attacking its reflection in a mirror on the browser

7. Run “./server 10003” on a terminal
8. Run “.client 192.168.0.151 10003” on a different terminal
9. Run “localhost:10003/ucla.jpg” on a browser (I used firefox)
* Displays a picture of UCLA's Powell Library on the browser

All the console outputs are listed in Sample outputs. An important thing to note is that TCP requires you to wait a certain amount of time for a port to free up before using it again. I was surprised at this but got a very good clarification from Seungbae regarding this.

Sample outputs:
Part A:
Here is the message: GET / HTTP/1.1
Host: localhost:10000
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:39.0) Gecko/20100101 Firefox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzi

Each field is defined as follows:
* The first line is an HTTP request line. This is the bare minimum HTTP request in this case because we are not specifying an HTML page.
* The second line is the beginning of the header. From here, we learn about the host, in this case “localhost”. We need this in case of virtual hosting, where a particular web server can host a wide variety of domains and websites. We also see our port number (10000).
* User-Agent provides information on the client software. We can see I am running Mozilla on Ubuntu with a Linux x86_64 machine.
* Accept: Describes what media/content is acceptable to the client. We can accept html, xml, and xhtml + xml files for this server. The q parameters after specifies which type of content is “preferred”.

* Accept-Language: This is most often used to compress data, indicating if you want to provide content in many different languages.
* Accept-Encoding: Refers to the type of encoding scheme that the server will accept, in this case gzi.

Part B:
1) test.txt
This is the HTTP request message from the client:
GET /test.txt HTTP/1.1
Host: localhost:10001
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:39.0) Gecko/20100101 Firefox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: __utma=111872281.1532816556.1454692221.1454692221.1454694844.2;
__utmc=111872281;
__utmz=111872281.1454692221.1.1.utmcsr=(direct)|utmccn=(direct)|utmcmd=(none)
Connection: keep-alive

* The test.txt file is the file we used the GET request on. The host is localhost at port 10001. A cookie is a small piece of data sent from a website and stored in the user's web browser while the user is browsing it. Our connection is in “keep-alive” mode because I have to manually stop the server using CTRL+C.

2) dog.gif
This is the HTTP request message from the client:
GET /dog.gif HTTP/1.1
Host: localhost:10002
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:39.0) Gecko/20100101 Firefox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: __utma=111872281.1532816556.1454692221.1454692221.1454694844.2;
__utmc=111872281;
__utmz=111872281.1454692221.1.1.utmcsr=(direct)|utmccn=(direct)|utmcmd=(none)
Connection: keep-alive

* The dog.gif file is the file we used the GET request on. The host is localhost at port 10002. A cookie is a small piece of data sent from a website and stored in the user's web browser while the user is browsing it. Our connection is in “keep-alive” mode because I have to manually stop the server using CTRL+C.

3) ucla.jpg
This is the HTTP request message from the client:
GET /ucla.jpg HTTP/1.1
Host: localhost:10003
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:39.0) Gecko/20100101 Firefox/39.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: __utma=111872281.1532816556.1454692221.1454692221.1454694844.2;
__utmc=111872281;
__utmz=111872281.1454692221.1.1.utmcsr=(direct)|utmccn=(direct)|utmcmd=(none)
Connection: keep-alive
* The ucla.jpg file is the file we used the GET request on. The host is localhost at port 10003. A cookie is a small piece of data sent from a website and stored in the user's web browser while the user is browsing it. Our connection is in “keep-alive” mode because I have to manually stop the server using CTRL+C.